PL/0 USER GUIDE

This PL/0 compiler is comprised of three modules: A scanner, a parser/code generator, and a virtual machine. These modules together take high-level PL/0 code, scan through it, and generate useable code that is read by a virtual machine to perform desired operations. The system should warn of any user created errors, such as with syntax, scoping, etc.

BASIC LINUX COMMAND LINE OPERATIONS

This compiler is intended to run from the command line. Here are some basic operations that will be useful to navigate to the PL/0 folder.

cd – Typing “cd” followed by a directory name will take the user into said directory, giving them access to all files contained within it.

ls- Lists all the files in the current directory

cd .. – This command will take the user into the directory one level above the current one.

COMPILING THE PL/0 MACHINE

First, navigate your way into the “pMachine” folder. Once inside, you will notice several source code files, such as “README.txt” and “Makefile.” This indicates you are in the correct place; at this point you should perform the following command:

* Type **make** and hit enter. This serves to compile the entire program.

RUNNING THE PL/0

Now that the pMachine has compiled, you must create or place an input file for the compiler. Create a file in this directory called “input.txt”, which should have your typed PL/0 code (more about this later). There are several options for running it. Each option will execute the pMachine and attempt to output the following:

* **cleaninput.txt**, a copy of the input code minus any comments.
* **lexemelist.txt**, which contains a list of all tokens found.
* **lexemetable.txt**, which is similar to the lexeme list, except it also provides the user’s original code with the matching tokens.
* **symlist.txt**, which provides a list of all symbols found in the code (symbols being identifiers, procedure names, and constant names).
* **stacktrace.txt**, a step-by-step diagram of the contents of the stack as code is run.
* **mcode.txt**, which contains code generated by the code generator

Any one of the following commands will run the PL/0:

./pMachine → Standard run. outputs the necessary files, checks for errors.

./pMachine –l → Standard run, but also prints the lexeme list to the screen.

./pMachine –a → Standard run, but also prints the mcode to the screen.

./pMachine –vm → Standard run, but also prints out the stack trace to the screen.

Upon completing execution, the program will list any errors found, and any extra content if the user prompted for it (i.e., the –l, -a, or –vm).

THE PL/0 LANGUAGE

PL/0 Syntax consists of high level code written in standard English. An example:

**var x, w;** //Declare variables x and w.

**begin** //begin indicates the start of actual code (besides variable declaration).

**x≔4;** //x becomes 4. This means the letter x references 4, and can be treated like a digit.

**w≔5;** //Similarly, w becomes 5. All references to w implicitly refer to the digit 5.

**if w>x then** //if w>x says “check if w (5) is greater than x (4). If true, then run the following.

**w≔ w + 1;** //if the “if” statement is true, w becomes w+1 (w≔5+1, so w≔6)

**else** //In the event the previous if statement was NOT true, run the following instead.

**w≔ x;** //w becomes x. Thus the variable w now represents whatever value x is.

**end.** //Marks the end of our block of code. The period indicates there is no code left to run.

Note the code can be essentially read like English and mathematics. “If this, then do this” is taken quite literally, and x≔4 is the same as x=4 in algebra. Also, all assignment statements should end with a semicolon.

OTHER USEFUL PL/0 COMMANDS

**procedure** x – Like a variable, it encapsulates the code that follows it until it reaches an **end**. All references to x (x can be any name) actually represents the code under the procedure.

**call** x – Runs the code encapsulated in procedure named x.

**write** x – displays variable x onto the screen.

**read** x – reads a value from user input, and stores it in x.

**while** condition **do** statement – As long as **condition** is true, the **statement** will run continuously. **Condition** refers to code that evaluates to either true or false (5>4, 4==4, etc.). A **statement** is a miscellaneous block of code and can be anything grammatically valid (add things, call a procedure, make a loop, etc.).

**odd** x – This is a conditional that will determine if x is odd.

FINAL NOTES ON PROCEDURES

Procedures are a very important part of any programming language, thus extra details are provided here to explain them.

A procedure is a way to compartmentalize code into a single, maneuverable unit. Imagine the entire program as a machine; then a procedure is a smaller machine within that machine, capable of being removed, inserted, shifted, etc. Procedures follow this format, in this order:

**(1) const declarations**

**(2) variable declarations**

**(3) sub procedures**

**(4) code**

Not every program needs all 4 of these segments, but variables and constants MUST be declared before use. As an example:

var f, n;

procedure fact;

var ans1;

begin

ans1:= n;

n:= n-1;

if n = 0 then f := 1;

if n > 0 then call fact;

f:=f\*ans1;

end;

begin

n:=3;

call fact;

write f;

end.

Take a look at **procedure fact;**. After that declaration, there is a variable declaration (**var ansi1**) and a block of code(the portion surrounded by **begin** and **end**). Thus we are following the model from before (procedure declaration, then constant and variable declarations, then subprocedure declarations, finally code). We chose to exclude any constant or subprocedure declarations, which is fine; only the relative order matters.

Once this procedure is created, it can be called like a mathematical function. Notice where we **call fact;**. This means the program will now run **procedure fact** at this exact location. It is as if the program replaced the call with a perfect clone of the procedure.

A final note: the entire program follows the same format as a procedure. Suppose this program is called FACTORIAL. We have variable declarations (var f,n), then sub-procedure declarations (**procedure fact** can be seen as a sub-procedure), and finally the code enclosed in **begin** and **end** (n≔3; call fact; write f;). We chose to leave out the procedure declaration (no need to say “procedure FACTORIAL”; it is implicit), and chose not to declare constants. The final **end** in the program must end with a period (**end.**), and the program is officially completed then.